UNIVERSITÀ DI PISA

WORkTogetHer

Progetto di Laboratorio di Reti

A.A 2020/21

Loris Giunta

matricola 560451 corso B

Indice

1. Introduzione a WORTH

WORkTogetHer è uno strumento per la gestione di progetti collaborativi che si ispira ad alcuni principi della metodologia “Kanban”.

Il sistema è costituito da un server che offre una serie di funzionalità e da clients che ne usufruiscono. La comunicazione tra server e clients avviene tramite 3 protocolli: TCP, UDP Multicast e RMI.

1.1 Metodologia Kanban

Una lavagna Kanban è uno strumento di gestione dei progetti agile progettato per aiutare a visualizzare il lavoro, limitare i lavori in corso e massimizzare l’efficienza (o il flusso). Le lavagne Kanban utilizzano cartellini, colonne e miglioramenti continui per aiutare i team e l’assistenza ad impegnarsi nella giusta quantità di lavoro. In questo modo risulta molto semplice monitorare visivamente lo stato e l’avanzamento delle attività. La flessibilità di Kanban consente di sovrapporla a flussi di lavoro, sistemi e processi esistenti senza interrompere ciò che è già stato fatto con successo.

2 Architettura Generale

L’architettura su cui si basa WORTH è di tipo client-server, dove il server fornisce servizi e i clients si collegano al server per usufruire di tali servizi.

Il server è realizzato multithreaded. Infatti, per ciascun client il server genera un Thread che si occuperà di gestire la comunicazione TCP con il singolo client.

2.1 RMI (Remote Method Invocation)

RMI è un’API che consente ad un oggetto di invocare un metodo su un oggetto che esiste in un altro spazio di indirizzi, che potrebbe dunque trovarsi sulla stessa o su una macchina remota.

Il server esporta due oggetti remoti: uno necessario al client affinché possa registrarsi a Worth, e uno affinché il client possa iscriversi e disiscriversi al servizio di callback offerto dal server. La callback permette al client di ricevere notifiche sulle registrazioni di nuovi utenti, cambiamenti di stato degli utenti (offline-online), eventuale rimozione di progetti e aggiunta di un nuovo membro ad un progetto.

Il server è in grado di notificare tali aggiornamenti, invocando i metodi dell’oggetto remoto esportato a sua volta dal client.

3 Server

La classe principale è ServerMain, che per l’appunto contiene il *main*. All’avvio vengono esportati gli oggetti remoti e ricostruiti il database degli utenti e quello dei progetti.

3.1 Persistenza dei dati

Il Server è in grado di mantenere persistenti i dati relativi agli utenti e ai progetti. Infatti, tutte queste informazioni vengono salvate in file .json, così da salvare lo stato dei 2 database e di poterlo ricostruire dopo la chiusura del server o un eventuale crash di quest'ultimo. Il tutto con l'ausilio della classe *Storage,* usata per scrivere e leggere i file .json.

Il Server genera una cartella *data* contenente a sua volta le cartelle: *UserDb* e *Projects.*

UserDb contiene il file *WorthUsers.json,* usato per la persistenza dello stato degli utenti

Projects contiene una cartella per ciascun progetto creato, e ognuno di questi progetti al suo interno mantiene una cartella con le *cards* e un file .json in cui sono salvate le informazioni sul progetto.

Per manipolare i dati contenuti nei file .json, quindi per deserializzare e serializzare, ho usato le librerie *Jackson* viste a lezione.

3.2 Tcp-Server

ServerMain contiene al suo interno l'oggetto *ServerTcp*, che per l'appunto si occupa di gestire le connessioni coi vari clients che lo richiedono. Il server rimane in attesa di nuove richieste (*Socket.acce*pt()) e una volta stabilita la connessione, un thread pool genererà un nuovo thread per gestire la comunicazione con il client.

Tale thread pool è di tipo *Cached Thread Pool***.** Questo crea nuovi thread in base alle esigenze, ma riutilizzerà i thread creati in precedenza se disponibili.

Al thread viene passato come task l'oggetto *TcpHandler,* la cui funzione è quella di handler, ovvero di gestire le richieste effettuate dal client al server.

La classe Tcp-Server inoltre avvia un ulteriore thread, che rimane dormiente sulla lettura dell'input. Infatti, nel caso di lettura della stringa "exit", il server chiude tutte le Socket Tcp attive, ed esegue lo shutdown del threadpool finchè tutti i task siano completati oppure all'occorrenza del timeout. Il server, dunque, prevede un comando per terminare la sua esecuzione, evitando di farlo in modo forzato.

3.3 Tcp-Handler

Un oggetto di questo tipo si occupa di gestire le richieste di un singolo client: l'handler si occuperà di portare a termine ciascuna richiesta e, una volta fatto ciò, restituirà una risposta a seconda se l'esito è positivo o negativo.

Per portare a termine le varie richieste, l'handler può accedere al database degli utenti e a quello dei progetti, cosa che invece il client non può fare: motivo per cui l'handler funge da intermediario tra il client e i database.

3.3.1 Interazione database-handler

Tutte le operazioni eseguite all'interno di entrambi i database sono thread-safe. Questo perché sono gli stessi database al loro interno ad effettuare accessi sicuri alle strutture dati in modo da evitare la race condition.

Infatti, nei vari metodi interni alle classi *UsersDatabase* e *ProjectsDatabase,* ci sono blocchi di codice sincronizzati tramite *syncronized(object){. . .}*

Ecco un esempio:

```
public ArrayList<UserStatus> loginDb(String nickName, String passw) throws
NullPointerException, UserNotFoundException, PasswException,
UserAlreadyConnectedException {

    if(nickName == null || passw == null)
        throw new NullPointerException();
    synchronized (users) {
        User u = users.get(nickName);
        if (u == null)
            throw new UserNotFoundException();
        if (!(u.getPassw().equals(passw)))
            throw new PasswException();

        if(u.getStatus().equals("online"))
            throw new UserAlreadyConnectedException();

        setUserStatus(nickName);
    }

    return getDbForClients();

}
```

Questa funzione è interna alla classe UsersDatabase, classe che per l'appunto si occupa di gestire le operazioni con il database degli utenti, rappresentata da *HashMap<String,User> users.* Essa contiene associazioni ***chiave:*** String nickName utente **=>** ***valore*****:** User utente, dove User è l'oggetto che definisce un utente.

Il metodo è usato per effettuare il login al sistema, previo controllo effettuato tramite programmazione difensiva, in modo da garantire il soddisfacimento delle precondizioni. Nel caso il controllo non vada a buon fine, verrà sollevata un'eccezione che sarà poi gestita direttamente dall'handler.

L'accesso all'HashMap viene effettuato in modo sicuro, sincronizzando il blocco di codice sull'HashMap stessa. Se non vengono sollevate eccezioni lo stato dell'utente passato come parametro della funzione, viene settato su "online".

3.3.2 Metodologia agile e Paradigma SOLID

L'espressione metodologia agile si riferisce a un insieme di metodi di sviluppo del software fondati su un insieme di principi comuni, direttamente o indirettamente derivati da principi del "Manifesto per lo sviluppo agile del software".

In questo stesso contesto è stato definito il paradigma *SOLID***,** che si riferisce ai primi cinque principi dello sviluppo del software e riassume in sé tutte le migliori pratiche per la programmazione orientata agli oggetti.

Seguendo questa linea, ho cercato di seguire alcuni di questi principi in particolare "Single Responsibility Principle" e "Information Hiding".

Single Responsibility Principle: consiste nel separare le varie responsabilità in modo che ciascuna classe ne abbia una ed una sola, incapsulata al suo interno.

Vediamo alcuni esempi:

- **UsersDatabase:** a questa classe ho delegato il compito di mantenere la struttura dati degli utenti iscritti a Worth, e tutte le operazioni di scrittura e lettura su questa struttura vengono eseguite all'interno di questa classe.
- **ProjectDatabase:** questa classe ha la responsabilità di mantenere la struttura dati dei progetti e di gestire tutte le operazioni relative a tale struttura.
- **Storage:** questa classe si occupa di effettuare tutte le operazioni di lettura e scrittura sui file .json. Permette il restore delle due strutture dati e la scrittura su json di eventuali modifiche a tali strutture. Inoltre, tiene traccia dell'ultimo indirizzo multicast usato per i progetti, in modo da poter ripartire dall'ultimo generato.

Information Hiding (incapsulamento): per motivi di sicurezza, ci sono oggetti che si vogliono rendere più sicuri e meno esposti di altri, in questo senso gioca un ruolo fondamentale questa tecnica di buona programmazione. L'incapsulamento consiste nel separare l'interfaccia dall'implementazione, cosicché i dati presenti all'interno possano essere modificati solo da funzioni interne all'oggetto stesso, e non da oggetti esterni. Un esempio è limitare l'uso dei metodi *getter e setter* all'interno delle classi, che se non necessari, possono portare oggetti esterni a modificare parametri potenzialmente "privati".

Ecco un esempio:

All'interno di UsersDatabase, la classe che contiene HashMap<String,User> users, la struttura dati degli utenti, non è presente alcun metodo getter e setter, poiché non voglio che altri oggetti esterni possano accedere alla struttura "privata". Questo mi garantisce di non esporre mai users, e che qualsiasi operazione rivolta ad essa, deve passare dai metodi interni a UsersDatabase.

L'uso di queste tecniche mi permette dunque di avere una gestione centralizzata delle responsabilità e della concorrenza, in modo che il chiamante non debba mai preoccuparsi di fare ciò.

4 Client

La classe principale che contiene il *main* è *ClientMain*, la quale si interfaccia con l'utente che richiede di accedere a Worth. All'avvio viene fatto il lookup dei due oggetti remoti esportati dal server, per la registrazione alla piattaforma e per l'iscrizione al servizio di callback. Il client a sua volta pubblica il suo oggetto remoto *RmiClientNotifyImpl* necessario al server per comunicare notifiche e aggiornamenti al client. Dopodiché il client effettua una richiesta di connessione Tcp al server e, una volta accettata, rimane in attesa sull'input per ricevere comandi da parte dell'utente, in loop finché non riceve il comando "close".

Il client è implementato in modo che possa rimanere in esecuzione anche quando un utente effettua il logout, così da poter ricevere ulteriori richieste di login. Il client però supporta un solo login alla volta; per poter effettuare un accesso con un altro utente, occorre prima eseguire il logout. Inoltre, non è possibile eseguire il comando "close" finché un utente è loggato: nel caso l'utente provasse a farlo, riceverà un avviso che lo invita prima a disconnettersi.

Dopo che viene letta la stringa in input, il client effettua uno scrupoloso *parsing* in modo da distinguere la richiesta da eventuali altri parametri. Se il comando richiesto dall'utente è corretto e presente tra i comandi disponibili in Worth, il client comunica via Tcp la richiesta al server. Solo nel caso dei comandi: "listUsers()" e "listOnlineUsers()" il client non esegue alcuna richiesta al server poiché mantiene in locale una sua copia del database degli utenti.

4.1 Database locale

Il client conserva una copia della struttura dati che contiene gli utenti. Tale struttura però non è esattamente all'originale: *LocalDb* è la classe contenente la struttura dati degli utenti posseduta dal client una volta che un utente effettua il login. Questa struttura è rappresentata da un *ArrayList<UserStatus> usersStatus*, dove UserStatus è un oggetto che contiene il nickname dell'utente e il suo stato (offline-online), senza far riferimento alla password. Le password difatti sono private e inaccessibili al client, altrimenti un qualunque utente loggato, potrebbe "conoscere" le password degli altri utenti iscritti a Worth.

Tale struttura dati viene aggiornata tramite Rmi-Callback dal server.

4.1.1 LocalDb e Callback

Ciascun utente loggato in un client, è in grado di ricevere in tempo reale aggiornamenti sul cambiamento di stato degli utenti e su eventuali nuove registrazioni al sistema. Tali aggiornamenti vengono notificati dal server al client invocando i metodi interni all'oggetto remoto esportato dal client, e le modifiche vengono apportate anche alla sua struttura dati locale.

La classe *UsersDatabase* che contiene la struttura dati principale, ha un metodo interno per trasformare i dati contenuti in *HashMap<String,User>* in dati accessibili al client, quindi senza le password.

```
private ArrayList<UserStatus> getDbForClients(){
    ArrayList<UserStatus> clientsDb = new ArrayList<>();
    Iterator<User> iterator = users.values().iterator();
    while(iterator.hasNext()){
        User user = iterator.next();
        UserStatus uS = new UserStatus(user.getNickName(),user.getStatus());
        clientsDb.add(uS);

    }

    return clientsDb;

}
```

con users di tipo *HashMap<String,User>*.

4.1.2 Funzionamento Callback

Vediamo un esempio nel dettaglio:

```
public synchronized void signUpForCallBack(RmiClientNotifyInterface
callBackClient, String idClient) throws RemoteException {
    if (!clients.containsKey(idClient)) {
        callBackClient.setIdClient(idClient); //associo il nome utente al nome
del client per la ricezione di callback
        clients.put(idClient, callBackClient);
        System.out.println("Utente " + idClient + " aggiunto al servizio di
notifica");

    } else {
        System.out.println("Utente " + idClient + " è gia iscritto al servizio
di notifica");
    }
}
```

Questo metodo è contenuto nella classe *RmiServerNotifyImpl.* Una volta effettuato il login, il client dell'utente che logga effettua la chiamata di questo metodo sull'oggetto remoto esportato dal server.

```
serverCallback.signUpForCallBack(stub, myargs[1]);
```

con *stub* = interfaccia dell'oggetto remoto esportato dal client e *myargs*[1] = nickName dell'utente

Dunque, se l'HashMap non contiene già la chiave idClient, vi aggiunge la nuova associazione. In questo modo il server tiene traccia di tutti gli utenti loggati nel sistema, in modo da potergli inviare notifiche e aggiornamenti.

5 Chat

Ciascun client, una volta che viene effettuato un login, istanzia un nuovo thread, al quale viene passato l'oggetto *MulticastChats*. Tale thread rimarrà in ascolto di nuovi messaggi, senza bloccare le altre operazioni effettuate dal client.

MulticastChats gestisce le chat di tutti i progetti a cui l'utente loggato partecipa utilizzando due strutture dati:

- **HashMap<String, InetAddress> infoProjects**: tale struttura contiene associazioni nomeProgetto -> indirizzo multicast di quel progetto
- **HashMap<String, ArrayList<Message>** contiene associazioni nomeProgetto -> lista di messaggi di quel progetto

Una volta che l'utente logga, viene effettuata la join sui canali multicast dei progetti a cui partecipa, e rimane in ascolto su ciascun canale.

5.1 Messaggio (lato sender)

Il messaggio da inviare è una stringa così definita:

nome del progetto + "\n" + utente che lo invia + "\n" + messaggio vero e proprio.

Tale messaggio viene inserito in un datagramma UDP e spedito all'indirizzo multicast associato al progetto.

Inserisco il nome di chi lo invia così da poter riconoscere se è un messaggio inviato da un utente oppure una notifica da parte del sistema.

Infatti, il sistema notifica aggiornamenti riguardanti l'aggiunta o lo spostamento di una card inviando un messaggio sul canale multicast del progetto considerato.

5.2 Messaggio (lato receiver)

Una volta ricevuto il datagramma UDP, viene creata una nuova stringa a partire dal risultato di *datagramPacket.getData().* A questo punto viene eseguito il parsing della stringa, separandola in sottostringhe in presenza del *carattere di nuova linea ("\n").*

Ottengo 3 sottostringhe: una col nome del progetto, una con il nome del sender e una con il messaggio.

Una volta fatto ciò, l'oggetto *Message(sender,messaggio)* viene aggiunto alla lista del progetto corrispondente.

In questo modo se l'utente richiede la lettura della chat di un determinato progetto, poiché ne conosce il nome, posso accedere in modo veloce alla lista dei messaggi di quel determinato progetto e stamparli a schermo.

5.3 Progetti e Callback

Il meccanismo di Rmi-Callback è utilizzato dal server anche per aggiornare l'utente su eventuali rimozioni di un progetto a cui partecipa oppure se viene aggiunto ad un progetto da un altro utente.

Il tutto viene effettuato tramite l'oggetto remoto *RmiServerNotifyImpl*, nel quale sono presenti due metodi per compiere tali operazioni.

Nel caso un utente venga aggiunto ad un progetto il client non è in grado di saperlo, allora sarà il server tramite Rmi a richiamare il metodo join sull'oggetto *MulticastChats* presente all'interno dell'oggetto remoto esportato dal client.

5.4 UDP Multicast

Le chat sono implementate secondo il protocollo **Udp multicast**. Ogni qualvolta viene creato un progetto, gli viene assegnato un indirizzo ip multicast, nell'intervallo *[224.0.1.0 – 239.255.255.255]*. In generale il range degli indirizzi multicast parte da *224.0.0.0*, ma ho riscontrato dei problemi dato che alcuni di essi sono riservati per determinati protocolli. Ho scelto di assegnare la stessa porta a ciascuna chat dei progetti.

Ho usato la classe *MultiGenerator* per appunto generare gli indirizzi, in particolare attraverso il metodo *generateIp()*.

Il sistema è pensato per ammettere il riuso: infatti la classe contiene una struttura dati (ArrayList) in cui salvare gli indirizzi ip dei progetti che sono stati cancellati. Nel caso in cui venga creato un nuovo progetto e la lista del riuso non è vuota, viene prima data priorità agli indirizzi all'interno di questa lista finchè questa non si sarà svuotata. La scelta dell'indirizzo da prendere da questa struttura è effettuata in modo casuale.

La classe Storage implementa anche dei metodi per rendere persistenti queste informazioni: infatti la cartella *data* contiene al suo interno il file *LastMultiAddress.json*, utile affinché il server sappia da quale indirizzo multicast ripartire, evitando così di sovrascrivere indirizzi ancora in uso.

6 Schema dei threads

Lato Server

- **Server Main Thread:** attivato all'esecuzione del server, si occupa di gestire le richieste di connessioni TCP da parte dei clients e di ricostruire lo stato delle strutture dati persistenti.
- **shutdownThread**: viene attivato nel Main Thread, e rimane in ascolto in background sull'input, in modo da poter terminare l'esecuzione del server.
- **Rmi Thread**: gestisce le funzionalità di registrazione e iscrizione al servizio di notifica via RMI
- **Cached Thead Pool:** il thread pool crea un thread per ciascuna connessione Tcp, di conseguenza in numero pari a quello dei client connessi al server.

Lato Client

- **Client Main Thread:** attivato all'esecuzione del client, gestisce la connessione tcp col server e le richieste da input dell'utente, rispondendo con gli esiti da parte del server.
- **Rmi Thread:** gestisce le funzionalità di ricezione di callback via RMI
- **MulticastChats Thread**: avviato nel Main Thead, rimane in ascolto sui canali multicast delle chat dei vari progetti, in modo da poter ricevere i messaggi.

7 Classi principali del programma

Ecco alcune delle classi principali usate nel programma:

- **ClientMain:** classe principale del client che contiene il main, implementa la CLI con cui l'utente può interagire, e stabilisce la connessione Tcp col server
- **LocalDb:** definisce il database locale del client
- **MulticastChats:** gestisce le connessioni multicast delle chat dei progetti di cui l'utente loggato fa parte
- **RmiClientNotifyImpl:** estende *RemoteObject* ed implementa l'interfaccia remota *RmiClientNotifyInterface.* Permette al client di ricevere via Rmi aggiornamenti e notifiche dal server
- **Card:** definisce la struttura di una card
- **Message:** definisce la struttura di un messaggio
- **User:** definisce la struttura di un utente
- **UserStatus:** definisce la struttura di un utente in modo limitato, con le sole informazioni accessibili dal client
- **Project:** definisce la struttura di un progetto
- **MultiGenerator:** usata per generare indirizzi multicast
- **ProjectsDatabase:** contiene la struttura dati dei progetti e implementa i metodi per manipolare tale struttura
- **UsersDatabase:** contiene la struttura dati degli utenti e implementa i metodi per manipolare tale struttura
- **RmiServerNotifyImpl:** estende *RemoteObject* ed implementa l'interfaccia remota *RmiServerNotifyInterface.* Permette al client di iscriversi al servizio di callback
- **RmiServerRegistrationImpl:** estende *RemoteObject* ed implementa l'interfaccia remota *RmiServerRegistrationInterface.* Permette al client di registrarsi al sistema
- **ServerMain:** classe principale del server contenente il main, esporta i due oggetti remoti, effettua il restore delle strutture persistenti e prepara il server Tcp
- **ServerTcp:** avvia un loop nel quale rimane in attesa di una nuova connessione tcp, e una volta stabilita, l'assegna ad un thread che si occuperà di gestirla tramite un handler.

- **ShutdownThread:** implementa *Runnable* e si occupa di rimanere in ascolto sull'input per un eventuale richiesta di chiusura del server. Se ciò accade, chiude le socket attive con i vari client, compresa la *serverSocket*
- **Storage:** classe che gestisce le operazioni di scrittura e lettura su disco, essenziale per garantire la persistenza dei dati.
- **TcpHandler:** implementa *Runnable* e si occupa di gestire tutte le richieste da parte del client, fornendo in risposta l'esito dell'operazione.

8 Istruzioni e comandi

Occorre prima avviare il server che, una volta effettuate le operazioni preliminari (restore degli utenti e dei progetti, esportazione degli oggetti remoti), mostrerà tale schermata:

```
Avvio del server Worth. . .


Server avviato
Digitare 'exit' per chiudere il server
In attesa di richieste...
```

A questo punto il server è in attesa di connessioni tcp ed è possibile avviare il client.

Una volta avviato, il client darà il "benvenuto" su WORTH e sarà possibile effettuare 3 operazioni inziali:

- Registrazione alla piattaforma
- Effettuare il login, solo se si è già registrati
- Chiudere il client

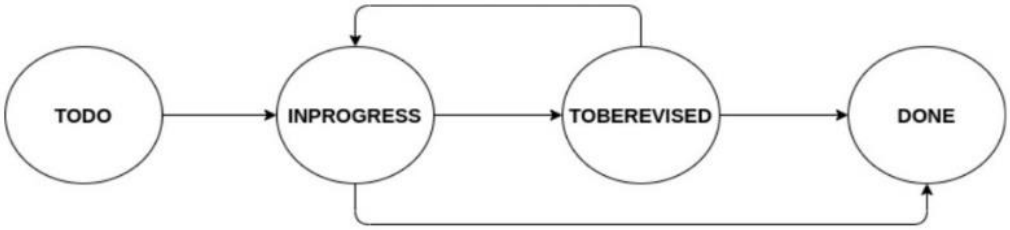
```
Benvenuto in WORTH
Prima di iniziare effettua il login se sei registrato, altrimenti registrati
Digita help se hai bisogno di supporto!

>
```

La CLI darà dei feedback su come interagire, ma nel caso in cui l'utente avesse dei problemi potrà sempre digitare il comando "help", che gli mostrerà tutti i comandi disponibili con la sintassi richiesta.

Ecco la lista dei comandi disponibili:

- **register username password:** registra un nuovo utente a WORTH
- **login username password:** effettua il login al sistema
- **logout username:** effettua il logout dal sistema
- **listUsers:** mostra la lista degli utenti iscritti a WORTH
- **listOnlineUsers:** mostra la lista degli utenti iscritti e online

- **listProjects:** mostra la lista dei progetti di cui l'utente che esegue il comando fa parte
- **createProject projectName:** crea un nuovo progetto
- **addMember projectName username:** aggiunge un nuovo membro ad un progetto
- **showMembers projectName:** mostra i membri di un progetto
- **showCards projectName:** mostra le cards contenute in un progetto
- **showCard projectName cardName:** mostra le informazioni di una determinata card di un progetto
- **addCard projectName cardName description:** aggiunge ad un progetto una nuova card con una breve descrizione
- **moveCard projectName cardName beginList endList:** sposta una card da una lista ad un'altra, secondo questo schema:



- **getCardHistory projectName cardName:** mostra la cronologia degli spostamenti di una card
- **readChat projectName:** mostra la chat relativa ad un determinato progetto, di cui l'utente fa parte
- **sendChatMsg projectName message:** invia un messaggio sulla chat di un progetto
- **cancelProject projectName:** elimina un Progetto, solo se l'utente ne ha i permessi e tutte le cards sono nella lista *DONE*
- **close:** termina l'esecuzione del client, possibile solo se non si è loggati

9 Compilazione ed esecuzione del programma

Vengono forniti 2 scripts all'interno della cartella *scripts*, utili per la compilazione ed esecuzione del server e del client sia su Windows che su Linux. All'interno ci sono gli script *compile_client.cmd* e *compile_server.cmd* per compilare rispettivamente client e server, e gli script *run_client.cmd* e *run_server.cmd* per eseguirli. Affinché il tutto vada a buon fine occorre trovarsi nella directory principale del progetto e lanciare gli scripts da lì.